An Investigation of Parallel Programming Techniques Applied to Monte Carlo Simulations for Post-Flight Reconstruction of Spacecraft Trajectory

R. Anthony Williams* and Justin S. Green†
NASA Langley Research Center, Hampton, Virginia, 23681

Parallelizing software to execute on multi-core central processing units (CPUs) and graphics processing units (GPUs) can be challenging. For some fields outside of Computer Science, this transition comes with new issues. For example, memory limitations can require modifications to code not initially developed to run on GPUs.

This work applies the Open Multi-Processing (OpenMP) and Open Accelerators (OpenACC) directive-based parallelization strategies on a Monte Carlo simulation approach for trajectory reconstruction enabling it to run on multi-core CPUs and GPUs. Large matrix operations are the most common use of GPUs, which are not present in this algorithm; however, the natural parallelism of independent trajectories in Monte Carlo simulations is exploited. Benchmarking data are presented comparing execution times of the software for single-thread CPUs, multi-thread CPUs with OpenMP, and multi-thread GPUs using OpenACC. These data were collected using nodes with Intel® Xeon® E5-2670 (Sandy Bridge) CPUs enhanced with NVIDIA® Tesla® K40 GPUs on the Pleiades Supercomputer cluster at the National Aeronautics and Space Administration (NASA) Ames Research Center (ARC) and a local Intel® Xeon Phi™ node at NASA Langley Research Center (LaRC).

Nomenclature

$A_0, A_1, A_2, B_0 B_1, B_2$	=	predictor constants
$a_0, a_1, a_2, b_{-1}, b_0, b_1, b_2$	=	corrector constants
Δt	=	time step size
n	=	time step number
y_n	=	variable of interest at time step n
$\dot{y}_n$	=	derivative of the variable of interest at time step n

I. Introduction

TRAJECTORY reconstruction is a process through which vehicle position, velocity, and orientation is determined post-flight. It is used to aid in the validation of pre-flight models and assist in identifying anomalies that may occur during flight. The fundamental approach to trajectory reconstruction uses the vehicle's initial state (position, velocity, and orientation), and integrates the inertial measurement unit (IMU) data to determine the vehicle states throughout its flight. Lugo et al.[1] developed a Monte Carlo based approach for trajectory reconstruction that incorporated the vehicle's final state information and introduces statistics. This method decreases uncertainties in the reconstruction results, which improves model validations and post-flight analysis. However, this Monte Carlo approach requires the integration of several thousand trajectories. These calculations are time consuming when executed serially, but the execution time can be decreased by utilizing concurrent computation.

* Doctoral Candidate, Dept. of Mathematics and Statistics, Old Dominion University, Norfolk, VA, 23529
Research Computer Scientist, High Performance Computing Incubator and Atmospheric Flight and Entry Systems Branch, NASA Langley Research Center, Hampton, VA, 23681.
† Doctoral Candidate, Dept. of Mechanical and Aerospace Engineering, University of Virginia, Charlottesville, VA, 22903.
Aerospace Engineer, Atmospheric Flight and Entry Systems Branch, NASA Langley Research Center, Hampton, VA, 23681.

The purpose of this work is to examine the use of parallel programming techniques on an algorithm that applies inertial navigation to trajectory reconstruction in a Monte Carlo dispersion process. The IMU errors and vehicle initial state conditions are distributed in a Monte Carlo sense using predetermined uncertainties. Therefore, each of the trajectories are independent of one another, enabling concurrent calculations using high performance computing (HPC) techniques and resources. This work utilizes HPC resources, such as multi-core central processing units (CPUs) and graphics processing units (GPUs), by implementing Open Multi-Processing (OpenMP) and Open Accelerators (OpenACC) directive-based parallel programming strategies.

An overview of the trajectory reconstruction software is presented in Section II. Section III provides specifications for the hardware that was used to collect benchmarking data. The parallel programming techniques used in this work, OpenMP and OpenACC, are discussed in Section IV. Compiler comparisons and execution times using OpenMP and OpenACC are listed in Section V. Benchmarking data are collected using resources on the National Aeronautics and Space Administration (NASA) Ames Research Center (ARC) Pleiades supercomputer cluster and a local node at NASA Langley Research Center (LaRC).

II. Trajectory Reconstruction Software

The software developed to conduct trajectory reconstruction is approximately 900 lines of C code. It begins by reading in parameter values from a data file, such as the radius and rotation rate for a given planet and the number of trajectories to be calculated. Then it reads the IMU data and the vehicle's initial state conditions that will be dispersed amongst the trajectories. Next, a loop iterates through the total number of trajectories. Within this section, each trajectory is provided the same IMU data to be integrated but its own initial conditions. There are two main routines called in the integrator function: gravity and an acceleration transformation routine. The gravity model used is a $J2$ model. The acceleration transformation routine receives acceleration data from the IMU, in the body coordinate frame, and it transforms the acceleration into the planet centered inertial frame. Once the first two time steps of the integration are calculated using the Euler method, the integration is completed in a large time loop using a numerical three-point predictor corrector scheme by Hamming[2]. The explicit method used for the predictor is defined by

$$y_{n+1} = A_0 y_n + A_1 y_{n-1} + A_2 y_{n-2} + \Delta t(B_0 \dot{y}_n + B_1 \dot{y}_{n-1} + B_2 \dot{y}_{n-2}) \tag{1}$$

where n represents the current time step, Δt is the time step size, $\dot{y}$ is the derivative with respect to time of y,

$$\begin{aligned} B_0 &= \frac{1}{12}(23 + 5A_1 + 4A_2) \\ B_1 &= \frac{1}{12}(-16 + 8A_1 + 16A_2) \\ B_2 &= \frac{1}{12}(5 - A_1 + 4A_2) \end{aligned} \tag{2}$$

and A_1 and A_2 are arbitrary constants such that $A_0 = 1 - A_1 - A_2$. In this work, $A_1 = -0.5$ and $A_2 = 0.5$. Once the predictor step is complete, each step is then updated by the corrector defined by

$$y_{n+1} = a_0 y_n + a_1 y_{n-1} + a_2 y_{n-2} + \frac{\Delta t}{24}(b_{-1}\dot{y}_{n+1} + b_0 \dot{y}_n + b_1 \dot{y}_{n-1} + b_2 \dot{y}_{n-2}) \tag{3}$$

where

$$\begin{aligned} b_{-1} &= 9 - a_1 \\ b_0 &= 19 + 13a_1 + 8a_2 \\ b_1 &= -5 + 13a_1 + 32a_2 \\ b_2 &= 1 - a_1 + 8_2 \end{aligned} \tag{4}$$

and a_1 and a_2 are arbitrary constants satisfying $a_0 = 1 - a_1 - a_2$. For this work, $a_1 = -0.5$ and $a_2 = 0.5$. Since each step of this integration is dependent on the previous three steps, this loop was not able to be parallelized. It is worth noting that this software originally executed with a MATLAB® wrapper that would determine which trajectories landed within some distance of the landing position. Statistical information would then be extracted from the remaining trajectories to create a set of normally distributed conditions for a new batch of trajectories to be reconstructed. A pseudo-code of the algorithm is provided in the appendix for reference.

III. Hardware Utilized

Execution time of the trajectory reconstruction software was collected using hardware on the Pleiades Supercomputer cluster at NASA Ames Research Center and a local Intel® Xeon Phi™ (Knights Landing or KNL) 7210 node at NASA LaRC. The nodes used on the Pleiades cluster have two Intel® Xeon® E5-2670 (Sandy Bridge) processors and one NVIDIA® Tesla® K40 GPU accelerator. Additional specifications on the hardware used for benchmarking are listed in Table 1.

Table 1. Hardware specifications[3-7].

	NASA Ames Pleiades Supercomputer		**NASA LaRC Node**
Hardware	Intel® Xeon® E5-2670 (Sandy Bridge)	NVIDIA® Tesla® K40	Intel® Xeon Phi™ 7210
Label Used in Paper	CPU	GPU	KNL
Release Year	2012	2013	2016
Number of Processor Cores	16	2880	64
Threads Per Core	2	1	4
Processor Base Speed [GHz]	2.6	0.745	1.3
Total L2 Cache [MB]	40	1.536	32
Memory Size [GB]	64	12	128
Max Memory Bandwidth [GB/s]	51.2	288	102

IV. Parallel Programming Techniques

The Monte Carlo based trajectory reconstruction process requires the integration of thousands of independent trajectories, and this *embarrassingly parallel* problem structure was exploited to implement parallel programming techniques such as OpenMP and OpenACC. These two techniques share many similarities in how they can be implemented, but differ in the hardware types there are typically used to target. Both OpenMP and OpenACC require compiler directives that divide work among parallel threads according to the architecture being targeted. Due to these differences, the amount of time and effort needed to parallelize this software using OpenMP was significantly less compared to OpenACC. Additionally, a modification was made to the original algorithm due to the limited amount of memory available when executing on a GPU. While the original algorithm provided a method of outputting the vehicle state conditions at each time step, the alteration to the algorithm reduced this output to only save the initial and terminal vehicle state conditions. Since the three-step numerical predictor corrector was used, only the previous three time steps are stored as temporary variables at each integration step to complete necessary calculations.

A. Open Multi-Processing

OpenMP is the technique used to parallelize this application using the CPU and KNL hardware. In this case, each trajectory is calculated on a separate thread that reads the initial state conditions and IMU errors for a particular case from global arrays of data. To enable this parallelization, the *#pragma omp parallel for* directive was placed above the trajectory loop to direct the compiler to parallelize the proceeding for loop. OpenMP produces a significant performance boost in execution time with a minimal amount of change to the code and need for hardware understanding, provided all of the memory is managed correctly. In this algorithm, for example, each thread needs to store vehicle state conditions to its own data array as they are calculated for a given trajectory. Thus the *private* data clause is invoked to provide each thread with its own copy of the array. Keeping data private to certain processing elements is similar when programming for multi-core CPUs and GPUs.

Additional work investigated the performance of this algorithm on KNL hardware. As shown in Table 1, the main differences between the CPU and KNL hardware is the total number of cores, the number of threads per core, and the

base processor speed for each core. A large advantage of the KNL is the high number of cores that it has while being able to execute the original algorithm without any modifications. The original algorithm stored vehicle state information at every time step, so the entire trajectory was written to disk. The difference in computation time between storing data at every time step and only storing the previous three steps was negligible. Furthermore, the main difference between the KNL OpenMP version and the CPU OpenMP version was an additional compilation flag (*-xmic-avx512*) to target the Many Integrated Core (MIC) architecture of the KNL.

Depending on the application and its workflow, the ideal number of threads to request varies[8]. For the CPU and KNL, an analysis was performed to determine the optimal thread count to use for this particular application under this particular configuration, and the results are shown in Figure 1. The curves identified in the legend are described as Hardware Type – Parallelization Strategy – Compiler. The number of threads used is set using an environment variable called *OMP_NUM_THREADS*. In the pgcc compiler version 17.1-0, used in this work, the maximum number of threads that can be used is limited to 64. The CPU and KNL have an optimal thread count equal to the total number of logical cores (number of cores × number of threads per core), which is 32 for the CPU and 256 for the KNL. The application execution speed increases linearly until the thread count used reaches the number of physical cores, then the additional speed up due to increasing the number of logical cores begins to plateau. The fastest execution time occurs when the thread count reaches the number of logical cores, but then begins to slow with increasing thread count. For the remainder of this work, all data reported for the CPU and KNL will reflect the use of 32 and 256 threads, respectively, as well as the modified algorithm saving off only the initial and terminal vehicle state conditions.

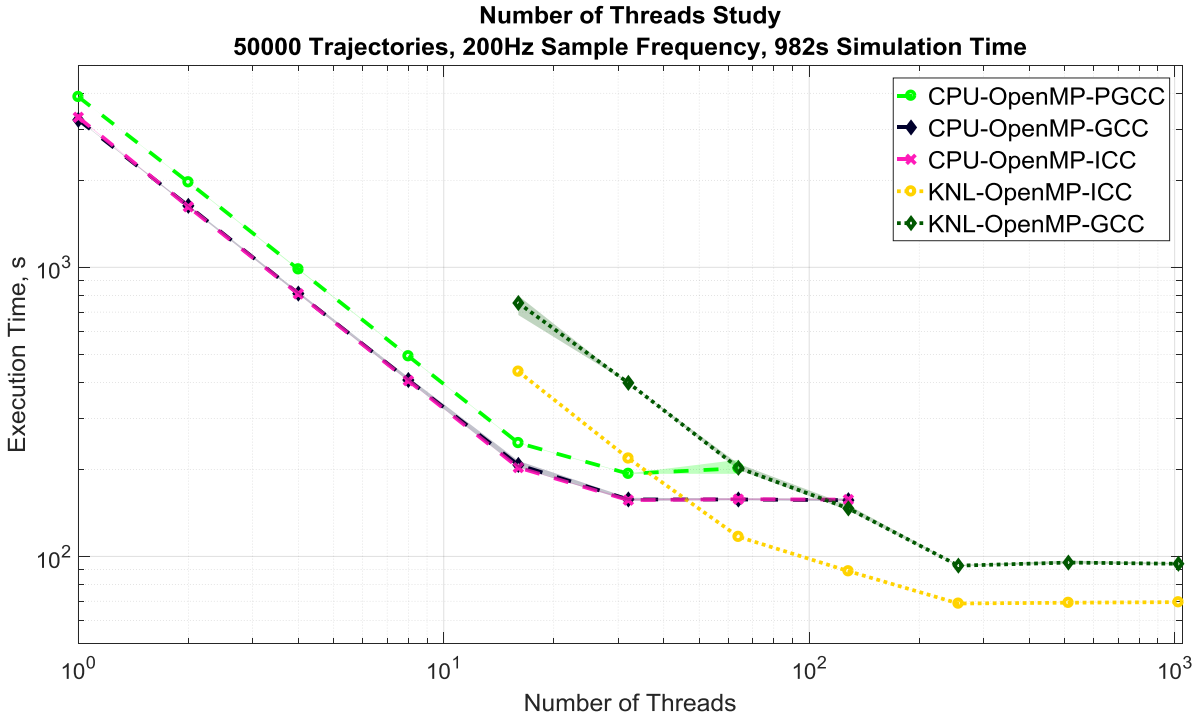

Figure 1: The effect of the number of concurrent threads on execution time. For each thread count, the application was executed ten times and the average execution time is represented by each mark. The shaded regions display the spread in the results.

B. Open Accelerators

A significant part of this work involved adapting this software to execute on a GPU, which was completed primarily over a five day intensive hackathon at NASA LaRC hosted by Oak Ridge National Laboratory. When working with NVIDIA GPUs, there are two main avenues for writing applications: develop the software using the Compute Unified Device Architecture (CUDA) parallel computing platform, or modify the existing software by adding OpenACC directives. CUDA is a low-level programming model that offloads much of the responsibility of memory management and workload mapping onto the programmer, whereas OpenACC is a higher-level programming interface that requires significantly less interaction from the programmer. The work presented here utilizes OpenACC, because it can be implemented in a similar manner as OpenMP. For example, the *private* data clause is used identically

between OpenMP and OpenACC. The activation of GPU parallelism using OpenACC is done using the directive statement *#pragma acc parallel loop*.

A common scientific computing application of GPUs incorporates large matrix operations[9]. The largest matrix operations for this reconstruction software involve 3×3 coordinate transformation matrices, which made the full utilization of GPUs a challenge. Memory management is another issue due to the limited memory on the GPU compared to the CPU, as seen in Table 1. Also, the time it takes to transfer data from the CPU (known as the host) to the GPU (known as the device), or vice versa, can slow the program execution speed. With this in mind, there are three main types of data that need to be considered: data transferred from the host to the device, data remaining on the device, and data transferred from the device to the host. In many cases, data are transferred from the host to the device and then back to the host once the computation is complete. For example, the data array that stores the vehicle state conditions is allocated on the host, transferred to the device to be written to, and then transferred back to the host. This array is approximately 15MB per trajectory since the state conditions of the vehicle are stored at every time step (approximately 2×10^5 total time steps), which requires large amounts of memory and limits the number of trajectories that can be calculated concurrently. Thus, an alteration was made to the algorithm to store only the initial and final states, which decreases the amount of memory needed for the vehicle state conditions array from 15MB to 240B per trajectory and enabled the code to run on a GPU. The downside of modifying the algorithm is that all of the intermediate state conditions are not saved to disk, but for the present analysis and proof-of-concept, initial and final states were sufficient.

Figure 2 shows the investigation into the optimal register* count per thread for this application. If each thread uses a small number of registers, then more threads can be active and execute code concurrently. However, if the memory needed to compute each trajectory reconstruction exceeds the amount of memory provided by the registers, then this leads to *spills* into local memory which results in a slowdown in program execution speed. The maximum number of registers to use per thread is set by the *maxregcount* compiler flag. For this particular application, with all variables defined as doubles (double precision), the optimal register count per thread is 255 (the maximum number per thread allowed by the hardware)[7]. With all variables defined as floats (single precision), the optimal register count per thread is 222. Though precision is lost when using floats instead of doubles, the relative error between the two cases is approximately 0.3% when comparing all vehicle final state condition variables except the z-component of the velocity. Since this value approaches zero, the absolute difference was examined and was equal to $0.17\ m/s$. For the remainder of this work, all data reported for the GPU float and double variable definition versions will reflect the use of 222 and 255 registers per thread, respectively, as well as the modified algorithm saving off only the initial and terminal vehicle state conditions.

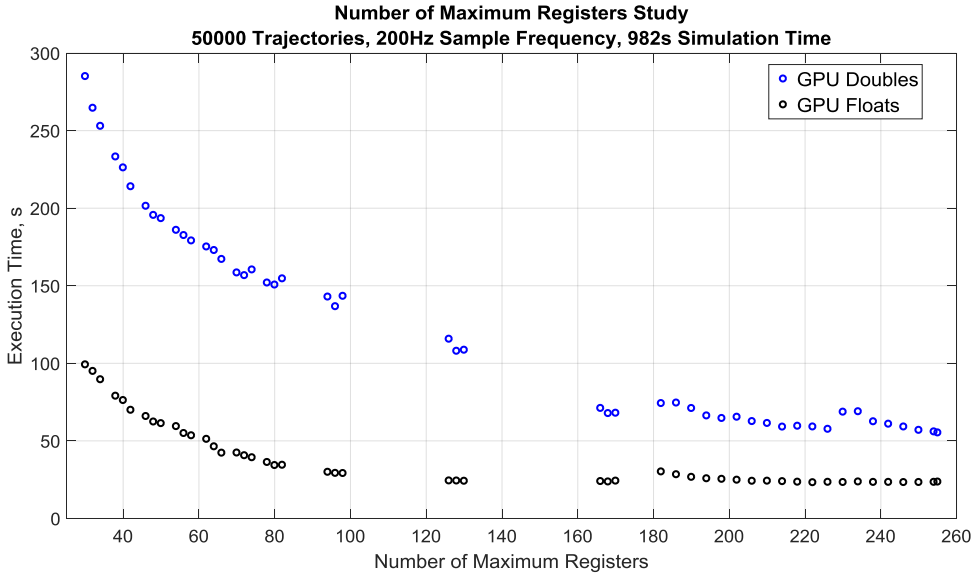

Figure 2: An experiment analyzing the effect the number of maximum registers provided at compile time has on execution time. For each register count, the application was run five times and the average execution time is represented by each mark. The spread for each average is not visible at this scale.

* In terms of speed, register files are the fastest type of memory on GPU devices[10].

V. Results

Comparisons of single-threaded and multi-threaded codes are made using three compilers: gcc (version 6.2.0), the GNU compiler; icc (version 18.0.0), the Intel C compiler; and pgcc (version 17.1-0), the C compiler from The Portland Group (PGI). Each compiler has different flags to apply optimizations, target specific hardware, and to enable OpenMP or OpenACC. All flags used for each compiler are listed in Table 2. At the time of this report, the gcc compiler on the Pleiades Supercomputer did not support certain OpenACC features, and thus pgcc was the only compiler used for the OpenACC on GPUs study. Additionally, the version of pgcc used was unable to compile for the KNL hardware.

Table 2. Compiler flags used to target hardware, enable optimization, and enable parallelism using OpenMP and OpenACC.

		GNU / gcc	Intel / icc	PGI / pgcc
Enable Optimization		-Ofast -flto -ffat-lto-objects	-fast -ffat-lto-objects	-fast -Minline -Mipa=fast,inline
Target Hardware	**CPU**	-march=native	-xhost	-
	KNL	-march=knl	-xmic-avx512	-
	GPU	-	-	-ta=tesla:cc35, maxregcount:130
Enable OpenMP		-fopenmp	-qopenmp	-mp
Enable OpenACC		-	-	-acc

A comparison of execution times for each hardware type and compiler combination is presented in Figure 3, where the curves identified in the legend are described as Number of Logical Cores – Hardware Type – Parallelization Strategy – Compiler. The number of reconstructed trajectories is scaled from 1000 to 50000 to examine the effect workload has on execution time. For the OpenMP implementation on the CPU and KNL hardware, the execution time scales approximately linearly with the number of trajectories reconstructed. On the CPU hardware, the Intel and GNU compilers produced similar results, while the PGI compiler execution time is slightly longer. A large distinction in execution time is seen between the CPU and KNL hardware implementations using OpenMP. Additionally, the software executed faster when compiled with icc compared to gcc, on the KNL hardware, for each trajectory count.

The GPU hardware performance varies depending on whether the variables used are defined as floats or doubles. The execution time when using all float variable definitions is approximately half of the all double variable definition version. However, for the CPU or KNL hardware the performance increase was less than 1% when using floats when compared to doubles. Additionally, the performance also varies depending on the number of trajectories that are being reconstructed. The execution time for the two GPU implementations remains approximately constant between 1000 and 2000 trajectories. Once the number of trajectories reaches 4000, the GPU execution time begins to scale approximately linearly with the number of trajectories. As the number of trajectories increases, the number of threads launched increases which can lead to an increase in performance for certain applications on the GPU[11].

Once the number of trajectories reaches 2000, the GPU hardware using all floats executes the fastest. However, if all doubles are used, the GPU only runs faster than the KNL once 16000 trajectories are being reconstructed. The GPU version of this application executes approximately 20% faster than the KNL when 50000 trajectories are used. If a larger number of trajectories is being reconstructed and the current trend in execution time continues, then the GPU will begin to outperform the KNL by a wider margin.

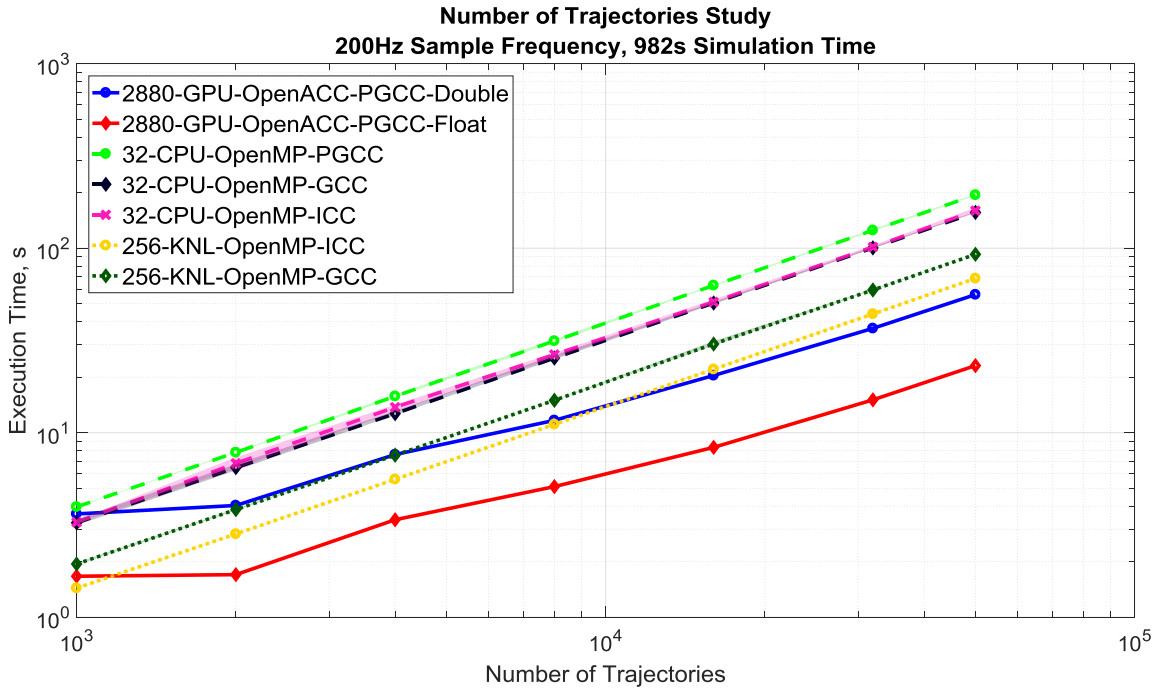

Figure 3: Comparisons of hardware and compilers while scaling the number of trajectories reconstructed from 1000 to 50000. For each trajectory count, the application was run 10 times and the average execution time is represented by each mark. The shaded regions display the spread in the results.

VI. Summary and Conclusions

This paper examines the use of parallel programming techniques on an algorithm that applies inertial navigation to trajectory reconstruction in a Monte Carlo dispersion process. The two parallel programming techniques being utilized are OpenMP and OpenACC, which are used on multi-core CPUs and GPUs, respectively. Two studies are conducted to determine optimal performance based on thread count with OpenMP and register per thread for OpenACC. Additionally, comparisons are shown between three different compilers and three different types of hardware.

For this particular application, the amount of time and effort to enable the software to run in parallel on GPUs was much more than to run on CPUs or KNLs. When all the variables being used are defined as floats, the GPU performs significantly faster than the KNL. Additionally, if the number of trajectories being reconstructed is large enough and all doubles are used, then the GPU will execute the software faster than the KNL as well. If the execution time trend continues as the trajectory count increases past 50000, then cost to benefit ratio would be more favorable to the GPU. However, if the number of trajectories being reconstructed is not large and the use of double precision variables is necessary, then the cost of adapting the software exceeds the benefit of utilizing the GPU. In this case, the KNL is the ideal hardware type to use. Though this result could be different for newer GPU hardware such as the NVIDIA P100 or V100, which will be tested in future work.

Appendix

A pseudocode for the trajectory reconstruction algorithm is presented below.

```
// Read data files
Read Parameter values from data file
Read IMU data and Initial conditions from data file

// Start timer
```

```
// Trajectory loop
// Initiate parallelization for both OpenMP (for CPUs/KNLs)
// and OpenACC (for GPUs)

#pragma omp parallel for private(trajectory_array)
#pragma acc parallel loop private(trajectory_array)
for (total number of trajectories) {

   Read row of IMU and initial condition array for the particular trajectory

   // Integration
   Given vehicle initial state, calculate first two time steps of IMU data
   integration using Euler method

   // Time loop, done sequentially
   for (number of time steps - 1) {
      Integrate IMU data to calculate (n + 1) time step according to numerical
      three-step predictor corrector method
   }

}

// End timer

// Output Data
Write initial and terminal vehicle state conditions for each trajectory to
data file
```

Acknowledgments

The authors would like to thank Dr. Rafael Lugo (Analytical Mechanics Associates) for his guidance and support, as well as the NASA LaRC High Performance Computing Incubator (HCPI) for the resources it provided. Resources supporting this work were provided by the NASA High-End Computing (HEC) Program through the NASA Advanced Supercomputing (NAS) Division at Ames Research Center. The 2017 ORNL Hackathon at NASA was a collaboration between and used resources of both the National Aeronautics and Space Administration and the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory. Oak Ridge National Laboratory is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Lugo, R. A., Tolson, R., Blanchard R., "Statistical Entry, Descent, and Landing Flight Reconstruction with Flush Air Data System Observations using Inertial Navigation and Monte Carlo Techniques", *AIAA Atmospheric Flight Mechanics Conference*, AIAA 2014-0388, National Harbor, 2014.

[2] Hamming, R. W., *Numerical Methods for Scientists and Engineers*, 2nd ed., Dover Publications, Inc., New York, NY, 1973.

[3] Dunbar, J., "Pleiades Supercomputer," *High-End Computing Capability* [online], https://www.nas.nasa.gov/hecc/resources/pleiades.html [retrieved 18 October 2017].

[4] "Intel® Xeon Phi™ Processor 7210," *Intel* [online], https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors/7210.html [retrieved 18 October 2017].

[5] "Intel® Xeon® Processor E5-2670," *Intel* [online], https://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI [retrieved 18 October 2017].

[6] "Tesla K40 GPU Accelerator: Board Specification," NVIDIA, BD-06902-001_v05, Nov. 2013.

[7] "NVIDIA's Next Generation CUDA™ Computer Architecture: Kepler™ GK110," NVIDIA, Whitepaper V1.0, 2012.

[8] Bienia, C., Kumar, S., Jaswinder, P. S., Kai, L., "The PARSEC Benchmark Suite: Characterization and Architectural Implications," 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, Toronto, ON, Canada, 2008, pp. 72-81.

[9] P. Maciol and K. Banas, "Testing Tesla architecture for scientific computing: The performance of matrix-vector product," *2008 International Multiconference on Computer Science and Information Technology*, Wisia, 2008, pp. 285-291.

[10] McKennon, J., "GPU Memory Types – Performance Comparison," *Microway* [online], https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison [retrieved 26 April 2018].

[11] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, Boston, MA, 2009, pp. 163-174.